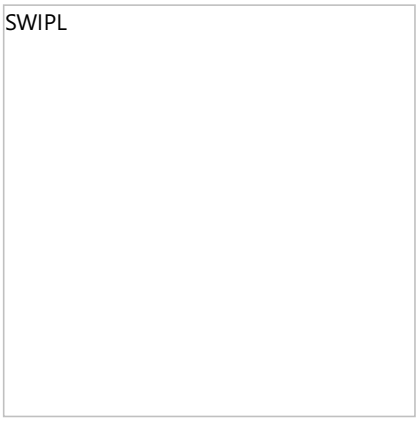
title: prolog 初探 date: 2017-10-06 14:51:52 tags: 编程语言



1. 前言

> 距离1972年--prolog正式诞生之年--已经过去很久了。在那个产生了众多明星语言的年代，prolog 以其独特的魅力脱颖而出，在人工智能领域大放光彩。然而，随着符号主义的衰落，prolog 也逐渐跌落神坛，少有问津，但是它仍然是一门伟大的语言，它解决约束满足问题的能力依然十分耀眼，它在专家系统、人工智能领域以及 jvm 中仍发挥着重要的作用。下面就让我们来初步了解一下这门语言。

2. 大纲

本篇博客的结论基于 swi-prolog 7.2.3 64位版，操作系统是 Ubuntu 16.04 LTS 版。按照我们学习一门编程语言的一般顺序，这篇博客首先会介绍 prolog 的编程范式，接着说明 prolog 的一些基础设施，比如数据类型、程序的组织方式、运行原理、基本操作、容器、基本技巧，最后通过解决一些小问题来进一步说明 prolog 的特点，最后贴一些介绍 prolog 进阶用法的文章的链接。要想在一篇博客中完整地覆盖一门编程语言是不现实的，因此文中提供了一些链接以供读者作深入了解。**这里需要指出的是本篇博客的正文部分不会包含如何安装、启动和配置 prolog 等内容。如果你像我一样，喜欢动手实践，请先阅读 附录： 如何安装并运行 prolog，它会指引你前进的方向。**

3. 声明式的逻辑编程语言，prolog

3.1. 声明式

与声明式相对应的编程范式是过程式。我们对过程式语言--比如 C 和 Java--已经很熟悉了。在过程式语言中，我们需要告诉计算机每一步应该怎么做。而在声明式语言中则不同，我们只需要告诉计算机我们想要什么，计算机就会把结果计算出来。声明式语言中使用最广泛的应该是 sql。在 sql 查询中，我们只需要描述自己想要什么样的结果，计算机就会从数据库中把结果找出来，在 prolog 中也是一样。比如，假设我们已经把如下的数据加载到内存中：

```
animal(cat).
animal(dog).
animal(fish).
```

输入如下查询：

```
?- animal(tomato).
```

结果如下：

```
?- animal(tomato).
false.
```

输入如下查询：

```
?- animal(X).
```

结果如下：

```
?- animal(X).
X = cat
```

如果我们想要更多结果，只需输入 **';'** 即可，如下：

```
?- animal(X).
X = cat ;
X = dog ;
X = fish .
```

在上面的例子中，我们用非常简单的代码实现了两个功能：

1. 如果输入的参数首字母是小写字母，那么如果参数是一种动物，输出 'true'，如果参数不是动物，则输出 'false'；
2. 如果输入的参数的首字母是大写字母，那么输出全部动物的名称。

我们会如何用命令式语言实现上面的简单功能呢？最容易想到的思路是用 if-else 语句或 switch-case 语句，根据输入的参数进行多次判断，以输出正确结果。这就与 prolog 的简洁方式大不相同了。在 prolog 中，我们只需要告诉解释器哪些是动物，查询时的其他工作都由 prolog 帮我们完成，代码就像自然语言一样容易理解。

3.2. 逻辑编程

prolog 的全称是 Programming in Logic，也就是说，你可以把逻辑语言(准确的说是一阶逻辑)毫不费力地转换成 prolog 程序。关于一阶逻辑的概念，我们不再赘述，事实上，我们完全不必知道一阶逻辑的概念也能学会 prolog 的粗浅用法，比如下面这个找出女明星的例子：

```
/*
# 以下排名不分先后
*/
star(迪丽热巴).
star(关晓彤).
star(白敬亭).
star(杨紫).
star(鹿晗).
girl(迪丽热巴).
girl(关晓彤).
girl(杨紫).
boy(白敬亭).
boy(鹿晗).

femaleStar(X):- girl(X),star(X).
```

前面若干行是数据库，这个我们已经在之前的例子里见过了，现在我们主要关注最后一行。我们要的结果是所有女明星的名字，那如何求这个结果呢？我们知道，如果一个人既是女性，又是明星，那这个人一定是女明星，那么我们可以得到如下规则：

```
A: 一个人是女性
B: 一个人是明星
C: 一个人是女明星

A AND B => C
```

在 prolog 中表示上面的逻辑关系需要用符号 **':-'** ，这个符号可以理解为如果，**"A:-B,C"** 可以理解为如果 **"B、C"** 均成立，那 **'A'** 成立。把规则用 prolog 的语法写出来，就会得到下面的代码：

```
femaleStar(X):- girl(X),star(X).
```

下面我们通过一些查询来验证代码的正确性：

```
?- femaleStar(X).
X = 迪丽热巴 ;
X = 关晓彤 ;
X = 杨紫.
```

可以看到我们的代码得到了所有的女明星的名字，下面再来一个例子：

```
?- femaleStar(鹿晗).
false.
?- femaleStar(杨紫).
true.
```

我们的代码正确的判断出了两位明星的性别。

3.3. 小结

通过上面的例子，你可能已经对 prolog 在声明式和逻辑编程方面的表现有了大致的印象，但是也有不少疑惑。在下面一节你的疑惑就会得到解答。

4. prolog 程序的语法、语义及基础设施

4.1. prolog 中的数据类型

prolog 的数据类型可以粗略地划分为四种：

1. 原子(atoms)：原子有三种，

- 第一种是由一对单引号包含的字符串，可以包含特殊字符,例如：'sdh*&' 这种用法与其他语言中的字符串字面量相同；
- 第二种是以小写英文字母开头并由大小写英文字母、数字、下划线组成，如 audi_a4，值得一提的是，在 prolog 解释器看来，第一种原子与第二种原子加上单引号后是相同的，也就是说，第二种原子不能被赋值,只能看作字符串字面量的语法糖;
- 第三种是一些特殊字符，如 ':-'，即操作符。

2. 数字(numbers)：即常数，如 4，5.0；
3. 变量(variables)：以下划线或大写英文字母开头并由大小写英文字母、数字、下划线组成；
4. 复杂项(complex terms)：由前面三种元素组合而成，如：

```
hide(X,father(father(father(butch))))
```

如果想更深入的了解 prolog 的数据类型，请参考 附录： prolog 中的数据类型

4.2. prolog 中的语句，仿函数

仿函数(functor)即形式类似函数的一种语法元素，是复杂项的一种。在 prolog 中，仿函数有名称和参数，但没有返回值，形式为

```
functorName(arg1, arg2, arg3, ...)
```

仿函数的名字一般是原子，参数可以是原子、常数和变量。prolog 的操作符也可以视为是特殊的仿函数。

4.3. prolog 程序的组织方式

从前面的例子中，我们可以看出，prolog 程序是由三部分组成的，这三部分的定义如下：

- 事实。事实是关于真实世界的基本断言。
- 规则。规则是关于真实世界中一些事实的推论。
- 查询。查询是关于真实世界的一个问题。

再结合 prolog 程序的数据类型和仿函数，我们可以给出 prolog 程序的一般形式：

- prolog 程序由事实、规则和查询构成，而事实、规则和查询都由仿函数构成，并且都以 **'.'** 结尾。
- 事实的一般形式是参数中不包含变量的仿函数，用于描述数据之间的关系；
- 规则的一般形式是由 **':-'** 连接的两部分，左半部分是一个仿函数，一般参数中会包含变量；右半部分是一串由 **','** 连接的被称为子目标的仿函数，各个仿函数之间是相与的关系，并且只有前面的仿函数的满足的情况下，才会对下一个仿函数进行匹配；
- 查询是由 **','** 连接的一串仿函数，当参数不包含变量时，查询结果为 true 或 false，当参数包含变量时，查询的结果为变量可能取的值。

4.4. prolog 程序的语义

到现在为止，我们对 prolog 的了解已经到了一定的程度，只需要再进行一点补充，我们就可以完全理解 prolog 语义了，接下来需要补充的就是 prolog 的核心概念：合一(Unification)。

合一与一阶逻辑有关，是一阶逻辑的归结过程中的重要工具。当然，你完全不用知道那到底是什么，你现在需要知道的是，'=' 是 prolog 中代表合一操作的符号，而且合一操作的作用是，找到一组映射，使得 '=' 两侧能够相互匹配，如果找不到，结果就是 'false'。比如说：

```
?- a = b.
```

结果肯定是 'false'，因为 'a' 和 'b' 都是原子，无法找到一组映射，使得两侧能够互相匹配； 再比如：

```
?- X = a.
```

结果就是：

```
?- X = a.
X = a.
```

这里能够成功匹配，因为 prolog 找到了一个映射使得两侧能相互匹配。这个映射当然就是：

```
X => a.
```

有了上面的两个例子，我们就可以给出合一的规则了：

> - 如果合一符号的参数是两个常量，那么只有两个常量相等时才能进行合一。
> - 如果合一符号的参数有一个是变量，那么变量被赋值为另一个参数。当两个参数都是变量时，一般的 prolog 的实现会把这两个变量指向同一个对象，但是具体如何处理还是要看实现。
> - 如果合一符号的参数两侧都是复杂项(就是两侧都是仿函数)，那么如果想要合一，首先这两个仿函数要有相同的名称和参数数量，它们位置相同的参数都得分别合一，并且参数的合一不能互相矛盾。

为了说明合一的最后一条规则，我们再举一个例子：

```
f(a).
f(b).

g(X, Y):- f(X), f(Y).
```

如果进行如下查询：

```
?- g(a, X) = g(X, b).
```

结果如下：

```
?- g(a, X) = g(X, b).
false.
```

可以看出，即使两个仿函数的位置相同的参数分别能够合一，但是由于参数的合一出现了矛盾，所以合一失败。

4.5. prolog 程序的运行原理

问题的复杂度总是存在，当一种语言的代码总是非常简单时，复杂度就转移到了编译器或者解释器中，prolog 正是这么一门语言。为了让我们只需要描述规则就能得到结果，prolog 解释器做了很多工作，其中的关键词是匹配和回溯，其中匹配的过程就是合一，同时也涉及到了一阶逻辑的归结与消解等内容；而回溯的搜索顺序则涉及到深度优先搜索和剪枝等内容。下面是 prolog 工作原理的简化描述：

> - prolog 以我们输入的查询的第一个子目标为起点对知识库从上到下进行匹配；
> - 找到可行的匹配后，如果匹配的是事实，则对下一个子目标进行匹配，如果匹配的是规则，那么从这个规则开始进行匹配。
> - 整个过程是按照深度优先的顺序进行的，当匹配失败时进行回溯(为了得到所有结果，匹配成功时也会回溯)，直至给出查询结果，或者说明查询是否可以满足。

前面我们提到过 **':-'** 可以理解为自然语言中的如果，这里需要注意的是，prolog 在搜索时是先搜索 **':-'** 左侧的部分，再搜索右侧的部分，这一点不难从 prolog 的运行原理中看出。

关于 prolog 的运行原理还有相当多的内容可以研究，比如搜索的顺序与规则中子目标的顺序的关系，回溯时的剪枝，如果想更加深入的了解 prolog 的运行原理，你可以前往 附录： prolog 运行原理 或者自行搜索相关论文。

4.6. prolog 程序中常用的技巧，递归

在其他语言中，我们经常使用迭代/循环来解决一些问题，但是 prolog 作为一门声明式语言，并没有提供相应的基础设施。所幸我们从 prolog 的运行原理中可以看出它天然支持模式匹配，这使得我们能够方便地构造递归来代替迭代。下面我们将通过一个例子来说明 prolog 中如何构造递归。

```
father(朱元璋, 朱标).
father(朱元璋, 朱棣).
father(朱标, 朱允炆).

ancestor(X, Y):- father(X, Y).
ancestor(X, Y):- father(X, Z), ancestor(Z, Y).
```

加载上面的知识库后，运行下述的查询：

```
ancestor(X, 朱允炆).
```

结果如下：

```
?- ancestor(X, 朱允炆).
X = 朱标 ;
X = 朱元璋 ;
false.
```

上面这个例子简明扼要，由于 prolog 支持模式匹配，我们很方便就对可能出现的两种情况采取了不同的处理方式，同时由于 prolog 会进行回溯，我们不用处理那些失败的情况，只需要关注成功的情况。相信对于任何一个掌握了一门通用编程语言的读者来说，prolog 中的递归都很容易理解。prolog 中使用递归就是这么简单，唯一需要考虑的就是时间复杂度的问题，有关如何提高递归的性能请自行搜索记忆化搜索与尾递归。

4.7. prolog 中的容器，元组与列表

prolog 中有两种容器，定长容器--元组，和变长容器--列表，下面是这两种容器的粗略定义：

> - 元组即用圆括号包围，以逗号分隔的一系列数据，其长度固定。
> - 列表则是用方括号包围，以逗号分隔的一系列数据，其长度不固定。
> - 列表与元组的元素不必是同一种元素

比如下面这两个例子：

```
# 列表
[[],  dead(z),  [2,  [b,  c]],  [],  Z,  [2,  [b,  c]]]
# 元组
([],  dead(z),  [2,  [b,  c]],  [],  Z,  [2,  [b,  c])
```

可以对列表和元组使用合一。下面是一个简单的例子：

```
?- (A,2,C)=(1,B,3).
A = 1,
C = 3,
B = 2.
```

```
?- [A,2,C]=[1,B,3].
A = 1,
C = 3,
B = 2.
```

可以看到，列表与容器作为复杂项完全符合前面所说的合一规则。但是合一在列表中又有不同用法，比如说下面这个例子：

```
?- [First,Second|Tail] = [1,2,3,4].
First = 1,
Second = 2,
Tail = [3, 4].

?- [First,Second|[Third|Tail]] = [1,2,3,4].
First = 1,
Second = 2,
Third = 3,
Tail = [4].
```

这个规则可以描述如下：

- '|' 把列表分成了两部分。
- 第一部分是列表的前几个元素，并以单个元素的形式出现。
- 第二部分是列表的剩余部分，以列表的形式出现。
- 在第二部分中，可以递归的使用这个规则。
- 这个特性只适用于非空列表。

列表的这个特性与递归结合，让我们可以用简洁的代码实现许多功能，比如列表拼接。prolog 已经内置了一个用于列表拼接的规则，示例如下：

```
?- append([1,2,3],[4,5,6],List3).
List3 = [1, 2, 3, 4, 5, 6].
```

我们现在就要实现这样一个规则。现在分析一下可能遇到的情况：

- 第一个参数是空列表，第二个参数是列表，则第三个参数应该与第二个参数相同。
- 第一个参数是非空列表，第二个参数是列表，则第三个参数应该由两部分组成，第一部分是第一个参数的头部，第二部分是一个新列表，这个新列表是第一个参数的尾部和第二个参数拼接的结果。

以下是代码：

```
app([], List, List).
app([Head|Tail], List, [Head|NewList]):-
app(Tail, List, NewList).
```

验证结果如下：

```
?- app([1,2,3],[4,5,6],List3).
List3 = [1, 2, 3, 4, 5, 6].
```

4.8. prolog 的内置规则

现在我们还剩下一些内容需要介绍，首先是一个内置变量：'_'，这个内置变量可以与任何数据进行匹配，一般用这个变量就意味着我们并不想了解与其进行合一的数据，比如我们想知道一个列表第二个元素是什么，我们可以这样写：

```
?- [_,X|_] = [1,2,3,4,5].
X = 2.
```

在上面的例子中，我们只关心列表的第二个元素是什么，并不关心其他位置的元素，因此，其他位置都用下划线进行匹配。

我们还需要了解一些内置规则，比如常见的符号，**'=='**，**'=='**，这两个符号是等于和不等于；**'write/1'**，输出一行；**'findall/3'** 用于查找所有答案；**'nl/0'**，换行。prolog 中还有一些用于 IO 的规则和其他一些有用的规则。

4.9. 小结

这一节中，我们主要介绍了 prolog 中的一些基础设施以及常用操作。这篇博客对 prolog 的介绍就到此为止了，下面的内容就是一些简单应用。

5. 用 prolog 解决一些小问题

5.1. 简化版四色问题

下面我们通过经典的地图着色问题来进行进一步的说明。 地图着色问题是给出一份地图，地图上相邻的国家不能用相同颜色印刷。这里我们指定颜色为红、绿、蓝三种颜色，并给出如下地图：

```
AAA
BCD
```

解决这个问题的思路很清晰，也很容易想到。当我们能从一个问题中抽象出事实和规则的时候，我们就可以用 prolog 解决这个问题，在这里我们先抽象出事实和规则和查询：

- 首先我们要定义事实，也就是颜色数据库；
- 其次我们要定义一个相邻规则，如果两个城市颜色不同，那么它们可以相邻；
- 最后我们要进行查询，内容为描述地图中的相邻关系，为了方便起见，我们把查询定义为一个规则。

代码如下：

```
% 首先定义颜色库
color(红色).
color(绿色).
color(蓝色).

% 再定义相邻规则
adjacent(X ,Y):-
color(X), color(Y), color(X) \== color(Y).

% 再定义查询
solve(A, B, C, D):- adjacent(A, B), adjacent(A, C), adjacent(A,D), adjacent(B, C), adjacent(C, D),
% 查询主体部分已经结束，下面是格式化输出
atomics_to_string([
    'A 是',   A,
    ', B 是', B,
    ', C 是', C,
    ', D 是', D
    ], Result),
write(Result), nl.
% 最后定义一个启动函数
go():-
findall(_, solve(A, B, C, D), _).
```

运行结果如下：

```
?- go().
```

```
A 是红色，B 是绿色，C 是蓝色，D 是绿色
A 是红色，B 是蓝色，C 是绿色，D 是蓝色
A 是绿色，B 是红色，C 是蓝色，D 是红色
A 是绿色，B 是蓝色，C 是红色，D 是蓝色
A 是蓝色，B 是红色，C 是绿色，D 是红色
A 是蓝色，B 是绿色，C 是红色，D 是绿色
true.
```

经过验证，各个方案都可行。

5.2. 常见面试题--列表翻转

翻转链表是程序员面试中出现的次数较多的一个简单问题，题目内容就和题目名称一样：输入一个列表，输出这个列表的倒序。这个问题如果用命令式语言来做其实是非常简单的：

> - 如果要翻转的是数组，那么只需要把位置对称的元素互换就可以得到其倒序；
> - 如果是链表也不难，只需要一次把子节点的指针指向父节点的指针即可。

在 prolog 中，列表对应的是数组。从前面的分析可以看出，无论是链表翻转还是数组翻转，这都是一个适合用循环去解决的问题。如果我们使用索引进行翻转，那么思路与命令式语言中相同，只是对于 prolog 来说，这种方式不像命令式语言那样简洁。既然我们已经了解了合一与 prolog 中列表的操作方式，为什么不试一下呢？

我们还是按照设计循环的思路来设计：

> 翻转列表肯定要有输入和输出，由于 prolog 中没有返回值，所以我们需要用一个参数来返回结果，现在不妨把输入定义为 **L**，把输出定义为 **R**， **R** 开始时是一个空列表。那么在翻转过程中，我们可以依次把 **L** 的元素放到 **R** 的头部；在翻转结束时， 输出 **R** 即可。
>
> 为什么要把 **L** 的元素依次放到 **R** 的头部呢？可以用下面的循环不变式来解释：
>
> > **L** 的头部元素按倒序恰在 **R** 的头部元素的前面。(有了这个循环不变式，我们就可以用数学归纳法证明这个思路的正确性。证明的工作这里不再详细描述。)
>
> 下面通过一个例子来说明：
>
> > 假如现在要翻转 [1,2,3]，那么我们首先调用 **([1,2,3],[])**， 把 **'1'** 放到一个空列表的头部，结果为 **([2,3],[1])**，这个时候，如果我们想得到正确结果，显然需要把 **'2'** 放到 **[1]** 的头部，此时的结果是 **([3],[2,1])**，此时，此时需要把 **'3'** 放到 **[2,1]** 的头部，经过此次调用，结果为 **([],[3,2,1])**，此时 prolog 会输出结果。

代码如下：

```
% 翻转过程中，依次把第一个参数的头部元素放到第二个的头部
cc([H|T],A):- cc(T,[H|A]).
% 翻转结束时，输出结果
cc([],A):- write(A).
```

效果如下：

```
?- cc([1,2,3,4],[]).
[4,3,2,1]
true.
```

为了调用方便，我们现在使用一个额外变量作为中间变量，最后用一个函数作为调用接口，更改后如下：

```
% 反转列表工具函数
accRev([H|T],A,R):-  accRev(T,[H|A],R).
accRev([],A,A).

% 反转列表接口函数
```

```
rev(L,R):-  accRev(L,[],R).
```

效果如下：

```
?- rev([1,2,3,4],X).
X = [4, 3, 2, 1].
```

5.3. 人工智能入门题--狼、羊、白菜问题

下面我们将解决一个经典问题，问题内容是：

> 一个人带着一只狼、一只羊和一篮白菜到了河边，想要去到对岸。他们唯一可用的交通工具是一艘船，显然只有人能驾驶船，并且船最多只能容纳两位乘客(白菜、狼、羊分别算作一位乘客)。在没有人监督时，狼会吃掉羊，羊也会吃掉白菜。那么问题来了，人如何在不遭受任何损失的情况下把狼、羊和白菜带到对岸？

这个问题乍一看很简单，只是求一个可行方案，实际上也很简单。在人工智能入门课上这种问题太多了，像八数码问题(类似华容道)、迷宫求解问题、寻路问题、野人-传教士问题，这些问题都可以用一个思路解决，那就是状态图搜索。而 prolog 的运行原理刚好是深度优先搜索，所以，prolog 非常适合解决这类问题。

现在大方向定了，只需要规定好一个可行的划分状态的方案就可以对状态图进行搜索了。那么如何规定状态呢？

> - 一个状态要能表示出我们解决问题所需的全部信息。我们需要知道各个乘客的位置，这就需要四个变量。我们还需要其他信息吗？其实很明显，不需要。因为我们只需要了解当前状态中乘客的位置就可以确定下一个状态有几种可能，只需要采取一定的搜索策略，比如 prolog 中的深度优先搜索，就可以知道当前路线是否可行。
> - 一个状态可以用一个元组表示，如下：
>
> ```
> (狼,羊,白菜,人)
> ```
>
> 元组从左到右的含义分别为狼、羊、白菜、人的位置，0为初始岸，1为目的岸，初始状态为(0,0,0,0)

确定了状态之后，我们需要能够重建路径。这个倒不难，可以在搜索的时候把当前路径中的状态都加入到一个列表中，最后打印列表就可以得到结果。现在思路非常清晰，已经可以考虑细节了。

既然是采用状态图搜索来解题，那么必须考虑状态直接如何跳转。根据题目规定，人渡河时一次只能带一个物品，那么人可以不带任何物品直接渡河，也可以带一件物品进行渡河。我们还需要检查转变之后状态是否有效，对状态的有效性的检查就可以从狼、羊、白菜的关系入手。如果狼和羊在一边而没有人监督，那么这一状态无效；同理，如果羊和白菜在一边而没有人监督，那么这一状态也无效。定义清楚之后，我们的代码几乎已经成型了:

```
% 无效判断
notValid((Wolf, Sheep, _, Man)):-
Wolf == Sheep, Wolf =\= Man.
notValid((_, Sheep, Cabbage, Man)):-
Sheep == Cabbage, Sheep =\= Man.

% 有效判断
valid(Term):-
not(notValid(Term)).

% 得到下一个状态
% 人带狼渡河
nextStep((Wolf, Sheep, Cabbage, Man), (Wolf1, Sheep, Cabbage, Man1)):-
Wolf == Man, Wolf1 is 1 - Wolf, Man1 is 1 - Man.
% 人带羊渡河
nextStep((Wolf, Sheep, Cabbage, Man), (Wolf, Sheep1, Cabbage, Man1)):-
Sheep == Man, Sheep1 is 1 - Sheep, Man1 is 1 - Man.
% 人带白菜渡河
nextStep((Wolf, Sheep, Cabbage, Man), (Wolf, Sheep, Cabbage1, Man1)):-
```

```
Cabbage == Man, Cabbage1 is 1 - Cabbage, Man1 is 1 - Man.
% 人独自过河
nextStep((Wolf, Sheep, Cabbage, Man), (Wolf, Sheep, Cabbage, Man1)):-
Man1 is 1 - Man.

% 得到下一个有效状态
nextValidStep(Old, New):-
nextStep(Old, New), valid(New).

% 狼、羊、白菜函数
% 得到结果时
wowo([(1,1,1,1)|Tail1], Result):-
Result = [(1,1,1,1)|Tail1].
% 搜索过程中
wowo([Head2|Tail2], Result):-
nextValidStep(Head2, NewHead), not(member(NewHead, [Head2|Tail2])), wowo([NewHead, Head2|Tail2],
Result).
```

细心的同学可能已经看出来最终得到的路径是倒序的，这时我们前面介绍的列表翻转函数就派上了用场。再加上输出结果的部分，添加的代码如下：

```
% 一个解决'狼-羊-白菜问题'的脚本
% 问题描述为：初始状态为人、狼、羊、白菜在河的一边，人想把狼、羊和白菜都带到对岸
% 但是人一次只能带一个物品，而如果没有人的看管，狼会吃羊，羊会吃白菜
% 问人如何把所有物品带到对岸
% 思路为使用一个长度为4的元组表示一个状态，元组从左到右的含义分别为狼、羊、白菜、人的位置，0为初始岸，1为目的岸
% 初始状态为(0,0,0,0)
% 每次人带一个物品到对岸，并根据带的物品更改状态，如果更改后状态有效，则从更更改后的状态继续寻找

% 反转列表工具函数
accRev([H|T],A,R):-  accRev(T,[H|A],R).
accRev([],A,A).

% 反转列表接口函数
rev(L,R):-  accRev(L,[],R).

% 主函数
solve(Answer):-
wowo([(0,0,0,0)], Result), rev(Result, Answer), write(Answer), nl.

go():-
findall(_, solve(Answer), Answer).
```

运行结果如下：

```
?- go().
[ (0,0,0,0), (0,1,0,1), (0,1,0,0), (1,1,0,1), (1,0,0,0), (1,0,1,1), (1,0,1,0), (1,1,1,1)]
[ (0,0,0,0), (0,1,0,1), (0,1,0,0), (0,1,1,1), (0,0,1,0), (1,0,1,1), (1,0,1,0), (1,1,1,1)]
true.
```

输出的结果略为简陋，但是可以看出，我们的结果是正确的。

5.4. 小结

这一节我们用 prolog 解决了几个小问题，其中地图着色问题的代码是我根据《七周七语言》书中的代码改的；列表翻转的代码最终版的来自于 列表翻转，demo 版是我根据链接中的代码改的；狼-羊-白菜问题的代码是我自己写的。

6. 结尾

prolog 虽然是一门受限严重的语言，但是漫长的时间使它逐渐变得全能，你现在可以用 prolog 开发 WEB 应用。然而作为一篇广告性质的文章，这篇博客介绍的东西其实非常有限，既没有把语言层面的东西介绍完，比如剪枝，也没有涉及到一些具体的应用，比如跨语言调用、I/O 库、多线程、协程以及上下文无关文法中的应用，这些东西就由读者自己去深入研究了。

7. 附录

7.1. swi-prolog 官方网站

官网首页| 官方文档| FAQ| 官网教程| 官网推荐书籍

7.2. 如何安装并运行 prolog

下载 prolog| 安装过程与安装普通软件无异，不再赘述| 运行 prolog| prolog 命令行参数| prolog 脚本编写与执行

7.3. prolog 语言相关

prolog 中的数据类型| prolog 语法| 合一| prolog 运行原理| 元组与列表| 内置规则与库与数学运算函数与操作符汇总